Expertise
and insight
for the future

Khanh Phan

Fullstack solution for two-sided market business model with containerization

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

20 April 2020

Author	Khanh Phan
Title	Developing a Shopping Aid System for Epidemic Situation
Number of Pages	33 pages
Date	20 April 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineer
Instructors	Janne Salonen, Head of School (ICT School)

The goal of the thesis was to create a prototype of a two-sided market as a platform for people to help each other during pandemic times. The secondary goal was to study web development with an alternative approach instead of the “JavaScript everywhere” MERN stack.

The application implementation was separated into two parts: An API provider in the backend with Django and a user interface with React. Various libraries and techniques were included in the process, notably Django Rest Framework, Docker and Travis CI.

Although most of the technologies are new for the author, the development process went smoothly, and a prototype was achieved. Further developments are recommended, but the prototype was sufficient as a two-sided market.

Keywords	Django, React, Docker, Fullstack development

Contents

List of Abbreviations

HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
XML	Extensible Markup Language
JSON	JavaScript Object Notation
UI	User Interface
UX	User Experience
DevOps	Development and Operations
CSS	Cascade Style Sheet
JSX	JavaScript XML
DOM	Document Object Model
MTV	Model Template View
MVC	Model View Controller
REST	Representational State Transfer
SQL	Structured Query Language
CI	Continuous Integration

1 Introduction

The year of 2020 has seen the outbreak of pandemic caused by the COVID-19 virus, and with how infectious the disease can be, many countries has gone into lock-down and thus the lives of people have been changed greatly. People are advised not to leave their homes unless it is necessary. In this case of pandemic, most of the people who contacted the virus will develop some mild symptoms and about of 80% of the patients will recover from the disease without hospital treatment.[1] However, there is still a higher risk group of people who are less resistant to the virus due to many reasons: Older people, young children, those with medical problems which may lead to complications, and those are immune-suppressed will have much more difficult time getting by longer quarantine duration. Because of that, and with many other unknown reasons, people will need help in getting the necessities for their daily life through online means due to social distancing.

The primary objective of the thesis is to develop a minimal viable product for a two-sided market as a potential solution for the aforementioned problem. The product acts as a platform which allows users to request and voluntarily provide supports during lock-down time. The secondary objective is to study web development and get familiar with the technologies used in the development and implementation of the application: The frontend is developed on React, Django for the back end with PostgreSQL as the database platform, containerized in Docker.

The thesis is divided into 5 parts. The first part includes the theories and background of full-stack development. The second part introduces the technologies which are intended to be used in the implementation. The third part shows examples of the implementation process with some discussions. The fourth part contains the author's personal take on the development process and the technologies used in it. Finally, in the conclusion of the thesis, summary of the thesis and how the application can be improved and used to solve the current problem for the pandemic.

2 Web Application

2.1 Definition, advantages and disadvantages of web applications

A web application is a software application which runs on web browser and utilizes web technologies to execute its features. Different from the native applications which need to be install in the computer and may gotten restricted to one platform/ operating system only, a web application is accessible through almost if not all platforms.[2] Given that every computer or mobile phone has a form of web browser, users are not required to install any new software to support running a web application. Web applications also has very little requirements of memory and computing power on the client since most of the heavy lifting are done by the server.

But that doesn't mean web applications have no drawbacks. Many browsers are different from each other. Browsers, as well as web applications are required to meet the code standards, even so each browser may behave differently to the same web application. Accessibility and availability are also a problem because of many factors such as: Connections from the client to the server, uptime of the server that hosts the application.

2.2 Common structure of a web application

A typical web application can be separated into two parts as sub-programs running simultaneously: Frontend and Backend.[3] Figure 1 shows the structure of a web application and how each component interacts with each other generally.

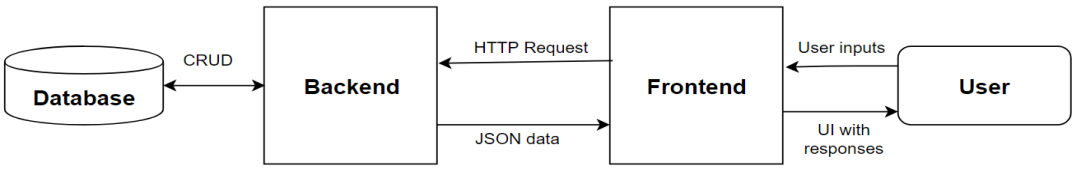

Figure 1. A typical web application component structure

Frontend (also called as Client-side) is the interface of the application, is what the users can see and interact with when they load the application on the browser. On the first load and every time the user makes a request, the Frontend will send a HTTP request to the server and displays the returned data properly.[4]

Backend (also called as Server-side) is the logic processor of the application, it provides the API for the frontend to make requests.[5] It also operates on the database in order to create new data or update, delete or fetch the current stored data, then process and return the data the frontend has asked for, commonly under XML or JSON format.

2.3 Web application development

As the structure of a web application are generally separated into two parts. The development accordingly consists of two main developments: Frontend development and backend development. There are also many side developments which can be found during or after the application development process, such as UI/UX designs or DevOps, but this thesis will focus on those two main roles.

Frontend development can be described as the process of implementing the interface of the application through coding. The development utilizes a mix of HTML, CSS and JavaScript [5]:

- HTML or Hyper Text Markup Language describe the structure of a web page using a series of elements and tags. The browser will use the description to render the web page.
- CSS or Cascade Style Sheet describe the layout and appearance of a web page by indicating how the HTML elements are on a web page, e.g. colors, font size, position…
- JavaScript is a programming language which gives web pages more functionality and interactivity.

Beside Vanilla JavaScript, several frameworks and libraries are being used these days for faster implementations and ease of complicated settings, for example: React for rendering web pages, D3.js for visualizing data, etc.

With the purpose of the interface is to be used by the users, having an UI/UX designer involved in the project is common practice in frontend development. The design/ mockup will act as a guideline for the developers during the coding process.

Backend development is the implementation of an application which runs in the back side of the web application. Its core usage consists of data operations on the database, logic execution and communication with the frontend on requests.[5] The number of server technologies provides a great deal of flexibility in choosing which direction the developers want to go with their server. Python, Java, C++, Node.js are some of the programming languages which can be chosen as the main language. For database technology choices, MySQL, PostgreSQL for SQL databases, or MongoDB for NoSQL databases. Other concerns of the backends consist of security, performance, deployment and maintenance are also crucial for the development.

Typically, in the planning phase of the project, a “stack” of technology for frontend and backend will be chosen. One of the oldest stacks is LAMP (Linux, Apache, MySQL and PHP) which gained its popularity since the 2000s, known for its maturity and stability.[6] Another commonly used stack is MEAN (MongoDB, Express.js, Angular and Node.js) with MERN (React instead of Angular) as an alternative, which enables developers to create a web application using JavaScript only.[7] Overall, it is developers’ choice to use a popular defined stack for better support or take on a new combination that may be more suitable to their needs.

3 Technologies and Tools

3.1 React

React is a JavaScript library for building the interface of a web application.[8] First released in 2013, it has been receiving frequent updates thanks to the combined effort from Facebook and the community. According to Stack Overflow's 2019 Developer Survey, React is one of the most used and loved web frameworks in the industry.[9]

The core of React is components. By breaking down the layout of UI into components following single-responsibility principle, which can be further broken down into smaller components, a hierarchy of components is achieved. This structure allows reusability of a common component, thus reduces the amount of coding needed. Figure 2 illustrates example of a Button component, which can be reused and customized through the usage of props.

```
const Button = props => {
  return (
    <div
      className={`${buttonStyles.btnContainer} ${
        buttonStyles[props.specialClass]
      }`}
      style={{
        height: props.height,
        width: props.width
      }}
      onClick={props.onClick}
    >
      <div
        className={buttonStyles.btn}
        style={{ backgroundImage: props.backgroundImage }}
      >
        <div className={buttonStyles.textContainer}>
          <p style={{ fontSize: `${props.textsize}` }}>{props.name}</p>
        </div>
      </div>
    </div>
  );
};
```

Figure 2. Code of a Button component, captured in Visual Studio Code

Each component is coded in JavaScript functions to declare the component and JavaScript XML (or JSX) for an abstract of its appearance on the UI. Then it will be created and inserted into the DOM, updated or removed when the data changes, following a lifecycle. This lifecycle of a component can be overridden to run specific codes using React's lifecycle methods. Figure 3 demonstrates the lifecycle of a component with common lifecycle methods.

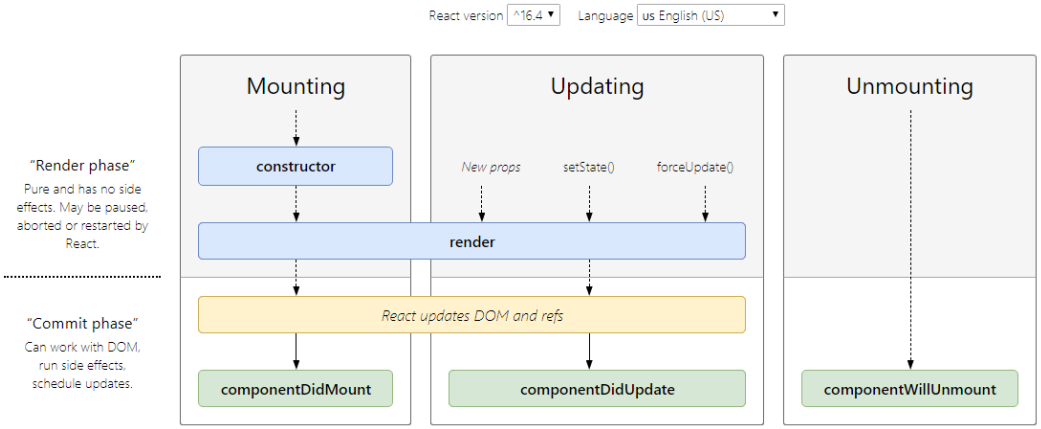

Figure 3. Lifecycle of a component with methods [10]

In the event of data changes, one or more elements will be changed, and the interface will need to be updated. A normal Browser DOM will then re-render the whole DOM tree, which may lead to performance issues once the interface grows in capacity and complexity. In React, when a component's state is changed, it will first compute the changes into a virtual DOM, then calculate the differences between the new virtual DOM and the previous virtual DOM (which called the "diffing" process). Finally, only the changed DOM objects will be updated in the real DOM. This approach results in a better performance than manipulating the DOM directly.[11]

As a View-focused JavaScript library, React requires extra libraries to complete the front-end's functionality. A few example libraries that often go with React are Redux or MobX for state management, React Router for routing pages, Material-UI or Ant-Design for styling components.

3.2 Django

Django is an open-source web framework written in Python, first released in 2005. It encourages fast development with clean design and simplifies the creation of a complex, database-oriented website.[12] According to Django's developers, it is designed on the philosophies of less coupling and more cohesion, less code, consistency and "Don't repeat yourself" principle. [13]

Django follows MTV (Model – Template – View) architectural design pattern. Many discussions were held about the right structure of Django, but overall it is accepted as a different interpretation of MVC (Model – View – Controller).[14] Model is the structure of data; it manages operations on data with the database. View describes how a web page and data will represent themselves to the user. Controller accepts inputs from the user and modify Model and View accordingly. The interactions are shown in Figure 4.

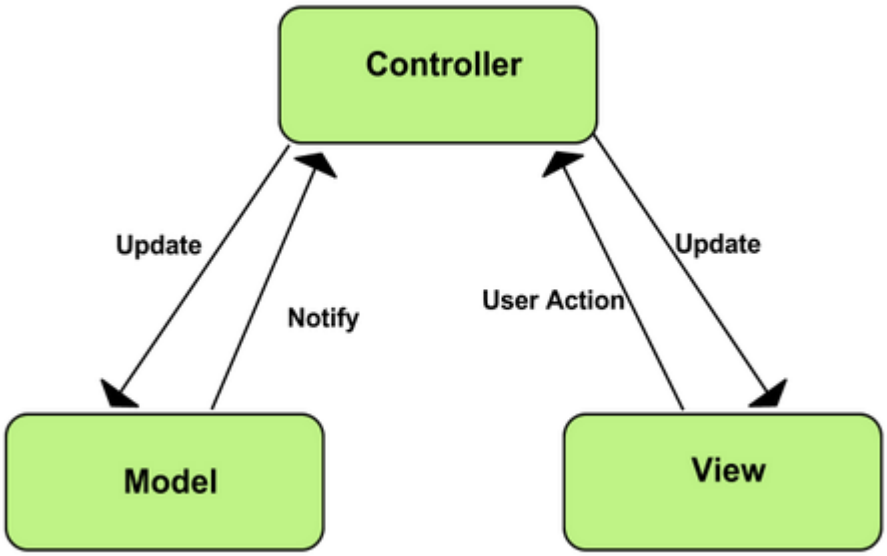

Figure 4. MVC architecture pattern [15]

In Django, Model is the same and Template is the View in MVC. View in MTV instead chooses which returned data is presented, and acts as a bridge between Model and Template. It is not necessarily the Controller in MVC, as the framework itself somewhat acts as a Controller. [16]

A typical Django project consists of one or several modules (app). Each module has an URL configured in the urls.py file. A module contains data model, template and view files. However, it is possible to move every data model into one module and let other models import the model instead, making the project easier to manage.

Even though Django has the capability of building the whole web application alone, there is an alternative approach of remodeling Django into a backend. It operates on the database and serve REST API to another framework which fulfills the role of frontend.

3.3 Django Rest Framework

While Django by itself supports developing REST API with request and response functions, it requires extensive customizations to get started. Because of that, developers often choose to include extra framework, commonly Django Rest Framework to help building the API. Django Rest Framework is an API building toolkit with Web browsable API, supported authentication policies and customized serialization.[17] It also provides many helper features, such as sets of generic views for quick end points generation or mixin classes for further configurations of basic view behaviors.

3.4 PostgreSQL

PostgreSQL is an open-source system for relational database management. Originally named as Postgres, it was developed in a project with the same name. In 1996, the project was renamed to the current name to reflect its support for SQL and has been getting frequent updates ever since.[18]

PostgreSQL is known for its architecture, reliability, data integrity, extensibility with a great deal of community effort in making innovative improvements. PostgreSQL is compatible with all major operating systems.[18]

Django is set up with SQLite configuration by default. While SQLite is an acceptable choice for small apps with quick local development and prototyping, it is recommended to use a more scalable database like PostgreSQL, to avoid the database-switching problem in the later part of development.[19]

3.5 Other Development Tools

3.5.1 Development on Virtual Environment with venv and Docker

The venv module in Python 3 allows the creation of virtual environments within their own directories. Each environment has its own version of Python, and any installed Python libraries and scripts are isolated from those installed in other virtual environment or a system Python.[20] This encapsulation helps the developers to reproduce the development environment without any potential conflicts with globally installed Python packages.

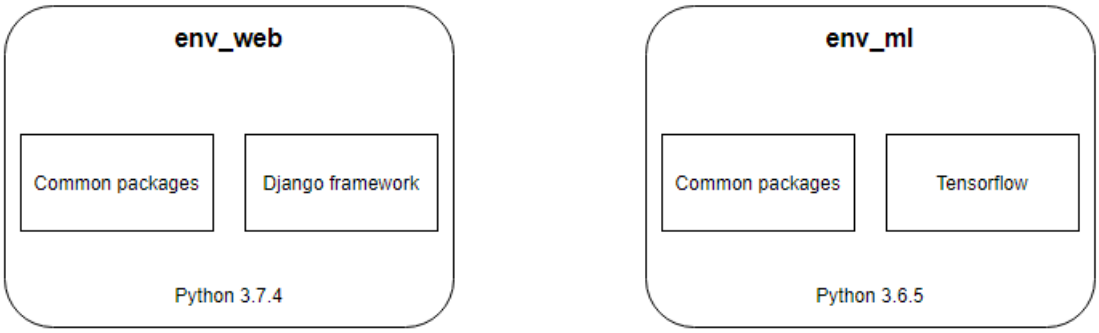

Figure 5. A Web project and a machine learning project using virtual environment

Figure 5 shows an example of two virtual environments. In this example, the developer can run two different versions of Python without having to reinstall on every switch, thanks to the virtualization.

Docker is a service product which bundles and delivers code and dependencies in packages called containers. The technology first designed for Linux but then receive a partnership with Microsoft and became available on multiple platforms.[21] Similar to how venv isolates the project directory and virtualize Python runtime, Docker isolates the whole application using OS-level virtualization. Docker is often compared to Virtual Machine as a less-demanding solution to virtualize applications. Figure 6 illustrates the differences in how containers and virtual machines work.

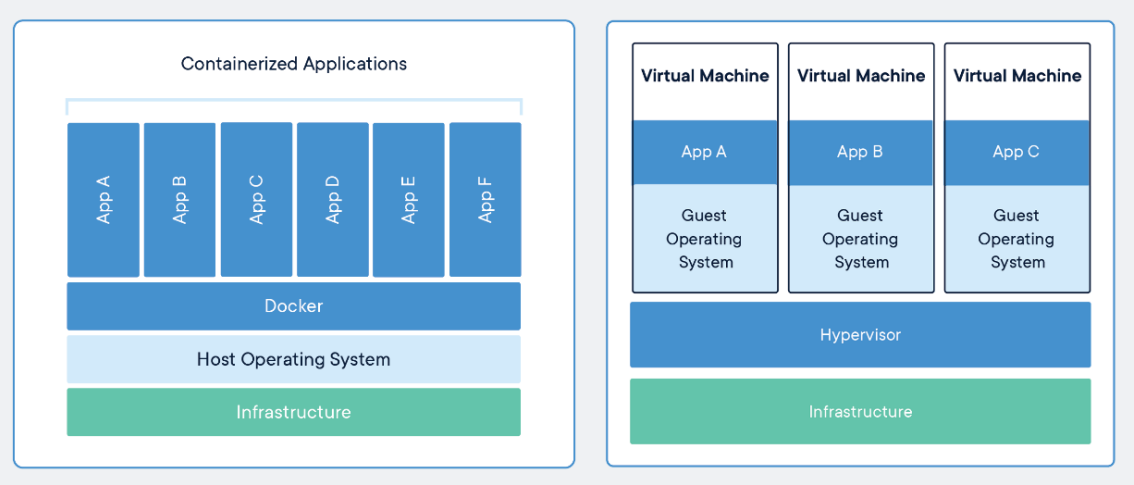

Figure 6. Virtual machines and containers architecture [21]

Virtual machines require a copy of an operating system with necessary libraries inside each machine and a Hypervisor to supervise all of them. Docker containers share the host operating system's kernel, take less memory and less time to deploy. Containers and virtual machines can be used together, offering a great deal of flexibility in deploying and managing applications. [21]

Configuration of a docker container can be done through Dockerfile. Many container presets for popular technology can be imported with tags, which are available on Docker Hub. Although using both Docker and venv is recommend, it is not necessary in this project since Docker will contain only a Python instance.

3.5.2 Version Control with Git

Git is a version control system, designed to keep track of changes in source code and enable collaboration between developers.[22] Every time the developer commit changes, Git will take a snapshot of the current system, then store a reference to that snapshot. All these snapshots will be stored in the local git folder, allows the developer to review the changes through multiple code version.

Git also has data storing and backing up capability, since it keeps snapshots of every files even if they get deleted in later versions and ensures their integrity.[22] Services utilizes Git to store data online are widely used by developers, notably GitHub, Gitlab or Bitbucket.

3.5.3 Continuous Integration with Travis CI

Continuous Integration is the practice of committing small code frequently, rather than committing a large change at once. Its goal is to get a healthier software development by developing and testing in smaller increments.[23] However, this requires more time deploying and repeating tests with high chance of passing. As a continuous integration platform, Travis CI takes care of this problem by automatically building and testing code changes, providing feedback on the outcome of the change. Travis CI also offers automatic deployment to application host if the new changes passed the tests.[23]

GitHub and Bitbucket directories are supported by Travis CI.[23] Users can choose to integrate Travis into all their directories or only specific ones. Every time the user pushes new code into the directory, Travis will start running the scripts as configured. This configuration is stored in the travis.yml file and can be changed according to the user's needs.

4 Project Implementation

4.1 Project overview

The goal of the project is to create a prototype of a two-sided marketplace web application. Since the project was started with the intent to help people during pandemic times, it contains these main features:

- Users are able to create their own accounts, edit their information, login from any devices that can connect to the application
- The user can create, update or delete his/her own requests when authenticated.
- The user cannot make changes to other users' requests.
- The user can view a list of current requests from other users, see the detail of a request, then choose to accept the request if suitable.
- The user can view the request sender's profile with contact information for further communications.

The project is implemented using technologies introduced in chapter 3. Users view and interact with the application through the interface made from mainly React, which can make requests using the API provided by the backend made from Django and Django Rest Framework. The implementation of front-end and back-end will be discussed in section 4.3 with examples to demonstrate the development process.

4.2 Development environment setups

4.2.1 Backend with Django

Although there are many time-saving options in setting up Django with related libraries, including project template like cookiecutter, the author chose to follow a tutorial on Docker documentation to gain more understanding about the process, with some changes for Windows operating system.[24] The setup starts with the configuration of the container using Dockerfile, a Python dependencies file and docker-compose.yml. An example of Dockerfile is demonstrated by Figure 7.

```
FROM python:3.7-alpine

ENV PYTHONUNBUFFERED 1

COPY ./requirements.txt /requirements.txt

RUN pip install -r /requirements.txt
RUN apk del .tmp-build-deps

RUN mkdir /app
WORKDIR /app
COPY ./app /app
```

Figure 7. A Dockerfile for Python 3.7 alpine

This Dockerfile is based on a Python 3.7 parent image. Then the dependencies defined in the requirements.txt are installed into the image. Finally, an app directory is created which contains the code of the application.

Once the Docker image is created, it can be run with Docker Compose. Docker-compose.yml is the setting for Docker Compose. It describes the services in the application, which image these services use, how they depend on together, and which port these service expose. Figure 8 shows two services named db and web in a docker-compose.yml file.

```
version: "3"

services:
    app:
        build:
            context: .
        ports:
            - "8000:8000"
        volumes:
            - ./app:/app
        command: >
            sh -c "python manage.py wait_for_db &&
                   python manage.py migrate &&
                   python manage.py runserver 0.0.0.0:8000"
        environment:
            - DB_HOST=db
            - DB_NAME=app
            - DB_USER=postgres
            - DB_PASS=dontdothis
        depends_on:
            - db


    db:
        image: postgres:10-alpine
        environment:
            - POSTGRES_DB=app
            - POSTGRES_USER=postgres
            - POSTGRES_PASSWORD=dontdothis
```

Figure 8. An example of docker-compose.yml

After that, a Django project is initialized with Docker Compose by running the command as follows:

```
docker-compose run web django-admin startproject regbackendapi .
```

This tells Compose to create the project regbackendapi in a container using “web” service’s image and configuration. However, since web image doesn’t exist yet, Docker Compose builds the image first with the setting in the file, then run the project creation command afterwards.[24] Finally, to connect PostgreSQL database to Django, the DATABASES field in regbackendapi/settings.py must be rewritten with the engine name for PostgreSQL along with the database configuration similar to Figure 9.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'HOST': os.environ.get('DB_HOST'),
        'NAME': os.environ.get('DB_NAME'),
        'USER': os.environ.get('DB_USER'),
        'PASSWORD': os.environ.get('DB_PASS'),
    }
}
```

Figure 9. Database settings for PostgreSQL

On Windows machines, an extra Python dependency named psycopg2 must be added into requirements.txt file before building the image. With the database setup finished, Docker Compose can run the container with the command as shown below:[24]

```
docker-compose up
```

Django then starts and runs at the port stated in docker-compose.yml file, finishes the backend environment setup.

4.2.2 Frontend with React

Setting up a React project can be effortlessly done with create-react-app template build tool. Only one command below is needed to build a new React app:

```
npx create-react-app my-react-app
```

npx is a node package runner, introduced in Node 5.2.0, allows users to run packages without having to install them globally.[25] Although the automatic installation may take a while, it helps skipping a long process of setting up webpack with babel to deliver a properly working React application. The development server then can be started in the application directory using the following command:

```
npm start
```

The application once deployed will automatically open a new browser tab directing to http:localhost:3000/ by default.[25] This is functional and development-ready, but as discussed in chapter 3, extra libraries are required to complete the frontend's functionality. To install libraries into the application, commands such as those in Figure 10 can be run.

```
npm install @material-ui/core
```

```
npm install -D jest
```

Figure 10. Node packages install commands

Figure 11. Package.json after installing the dependencies

The first command in Figure 10 shows the installation of material-ui, a library of styled react components. The second command installs Jest with the option “-D”, which indicates that Jest is development-only dependency. A dev dependency is only available in development environment and will not be included once the application is deployed to production. Both dependencies are register into Package.json under different object, as showed in Figure 11.

4.2.3 Automatic testing

In this project, the automatic testing is done by Travis CI with the GitHub. Once pushed to GitHub, the application directories are found on Travis CI dashboard. Figure 12 demonstrates an example of two directories on the dashboard.

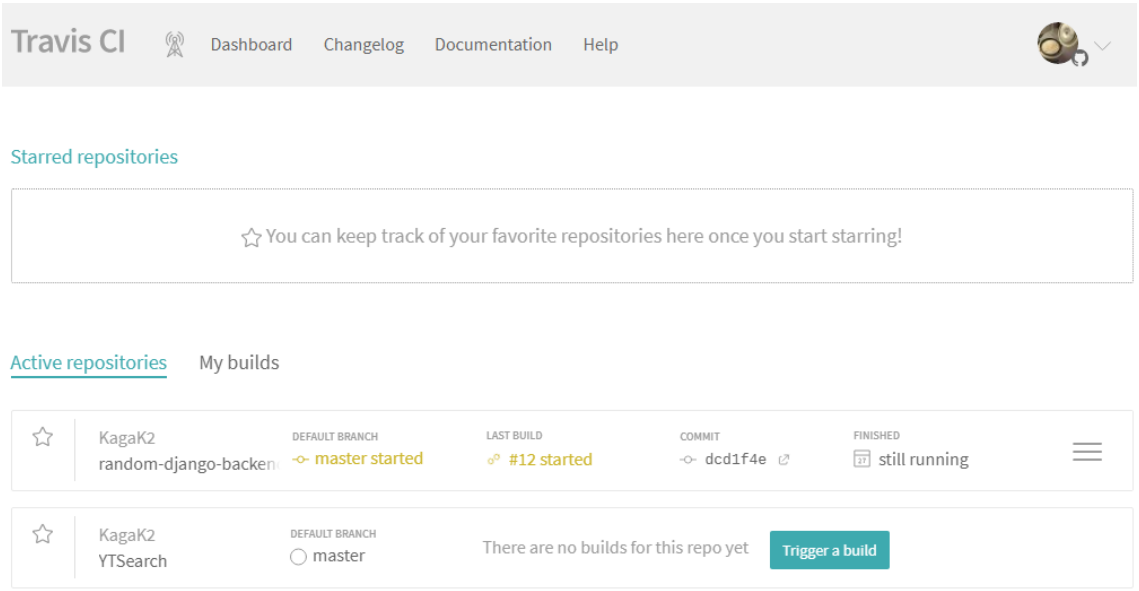

Figure 12. Travis CI dashboard

User can start the automatic build on Travis by putting a travis.xml file into the app folder. This file describes the build configuration with the script that Travis CI runs on every deployment. Figure 13 and 14 gives examples of travis.yml for Python and React application respectively.

```
language: python
python:
    - "3.7"

services:
    - docker

before_script: pip install docker-compose

script:
    - docker-compose run app sh -c "python manage.py test"
```

Figure 13. Travis.yml for Python project

```
language: node_js
node_js:
    - node

before_script:
    - npm install

script:
    - npm run test
```

Figure 14. Travis.yml for React/JavaScript project

After adding the Travis.yml file, Travis starts the build and runs the tests on every new push on the GitHub directory.

4.3 Project implementation examples

4.3.1 User model on Django

The user model will be implemented in the core app as the app to centralize every model in the project. The serializer with views is implemented in user app. Both apps are created in Django project using the following commands in the project directory:

```
docker-compose run --rm web sh -c "python manage.py startapp core"
docker-compose run --rm web sh -c "python manage.py startapp user"
```

The commands will create two directories named core and user with all common files for an app inside. However, these app are not linked to the project yet. An app must be defined in the settings.py of the main project to start working. (see Figure 15 below)

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'rest_framework.authtoken',
    'core',
    'user',
    'requests',
```

Figure 15. Application definition in settings.py

User model creation with authentication can be done painlessly using the custom user model. A custom user model inherits the base user model and overrides defined functions for easier implementation. Figure 16 shows a custom user model based on AbstractBaseUser model in core app, along with a custom UserManager.

```
from django.db import models
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager, Permis-sionsMixin

class UserManager(BaseUserManager):

    def create_user(self, email, password=None, **extra_fields):

        if not email:
            raise ValueError('Users must have an email address')
        user = self.model(email=self.normalize_email(email), **extra_fields)
        user.set_password(password)
        user.save(using=self._db)

        return user

    def create_superuser(self, email, password):

        user = self.create_user(email, password)
        user.is_staff = True
        user.is_superuser = True
        user.save(using=self._db)

        return user

class User(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(max_length=255, unique=True)
    name = models.CharField(max_length=255)
    contact = models.CharField(max_length=255)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = UserManager()

    USERNAME_FIELD = 'email'

    def __str__(self):
        return self.email
```

Figure 16. Custom user model and user manager in models.py

The User class contains many database fields such as email, name or is_active. Each of the fields has its own type and property. For example, is_staff field is a Boolean field, its value is either True or False, and set to False by default. In the BaseUserManager class, Django allows changing the default USERNAME_FIELD to another unique field. In Figure 16 the USERNAME_FIELD is set to “email”, enable users to login with their registered emails.

A custom user manager is also a common practice whenever a custom user model is used. It extends BaseUserManager with two additional methods: create_user and create_superuser. A special note in the UserManager class is the usage of a utility function named normalize_email to normalize the domain name. This function is included in the BaseUserManager class, along with other utility functions such as get_by_natural_key and make_random_password.

Django is configured with the default user model by default. To apply the new customer user model, the following code must be added to settings.py

```
AUTH_USER_MODEL = 'core.User'
```

A created model can be added to the admin page for management. Figure 17 contains the code for model registration with the result demonstrated in Figure 18.

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.utils.translation import gettext as _

from core import models

class UserAdmin(BaseUserAdmin):
    ordering = ['id']
    list_display = ['email', 'name', 'contact']
    fieldsets = (
        (None, {'fields': ('email', 'password')}),
        (_('Personal Info'), {'fields': ('name','contact')}),
        (
            _('Permissions'),
            {
                'fields': ('is_active', 'is_staff', 'is_superuser')
            }
        ),
        (_('Important dates'), {'fields': ('last_login',)})
    )
    add_fieldsets = (
        (None, {
            'classes': ('wide',),
            'fields': ('email', 'password1', 'password2')
        }),
    )

admin.site.register(models.User, UserAdmin)
admin.site.register(models.Tag)
admin.site.register(models.Ingredient)
admin.site.register(models.Recipe)
```

Figure 17. Custom user model added to admin page

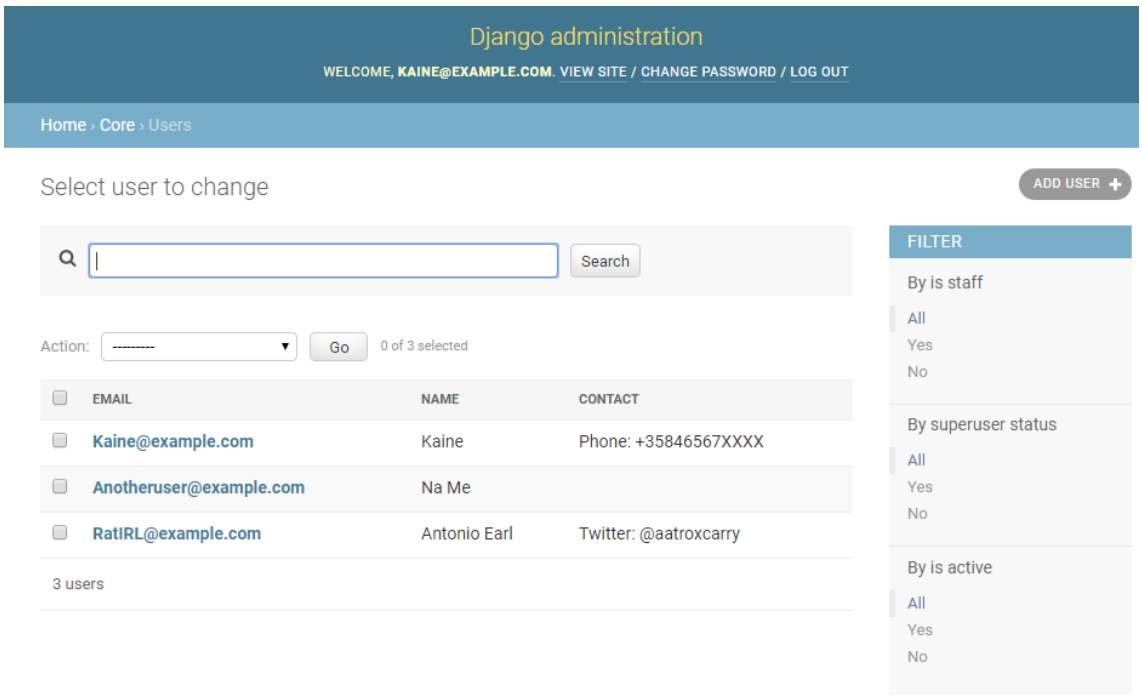

Figure 18. Django admin page with the list of users

4.3.2 Django serializers with authentication

For the implementation of the authentication system, there are many concerns about how Django and React should be connected. There are three common approaches which are widely used [26]:

- Approach 1: A single Django app to load a single HTML template for React to operate on
- Approach 2: Django and React will work separately. One provides the API and the other takes the API and runs as a standalone application.
- Approach 3: Multiple React apps for every Django templates.

The author to use approach 2 with the intent to separate the development of both side from the start. Approach 1 is easier to implement but dropped but due to the author's inexperience in setting up Docker with both Python and React. Furthermore, the second approach involves the usage of JWT and CORS headers, which is deemed more challenging and interesting for the author to study.

The setting for JWT and CORS in Django is done by installing and including the necessary libraries into settings.py. Figure 19 shows the configuration in the settings file.

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'corsheaders',
]

MIDDLEWARE = [
    ...
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    ...
]

...

REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_jwt.authentication.JSONWebTokenAuthentication',
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.BasicAuthentication',
    ),
}

...


CORS_ORIGIN_WHITELIST = (
    'localhost:3000',
)
```

Figure 19. corsheaders with rest_framework_jwtsettings

In the setting, CORS middleware must be place above Django's common middle to take priority in generating response. Then the Django rest framework config allows JWT to be the primary authentication method. Finally, whitelisting localhost:3000 enables local React development to make API calls to this server.

A view that provides a POST method for getting JWT can be added in urls.py (see Figure 20 below)[27]. This view can be used on the web browser, which returns a token after logging in successfully, as in Figure 21.

```
from rest_framework_jwt.views import obtain_jwt_token

urlpatterns = [
    #...
    path('token-auth/', obtain_jwt_token)
]
```

Figure 20. Enable obtain_jwt_token in urls.py

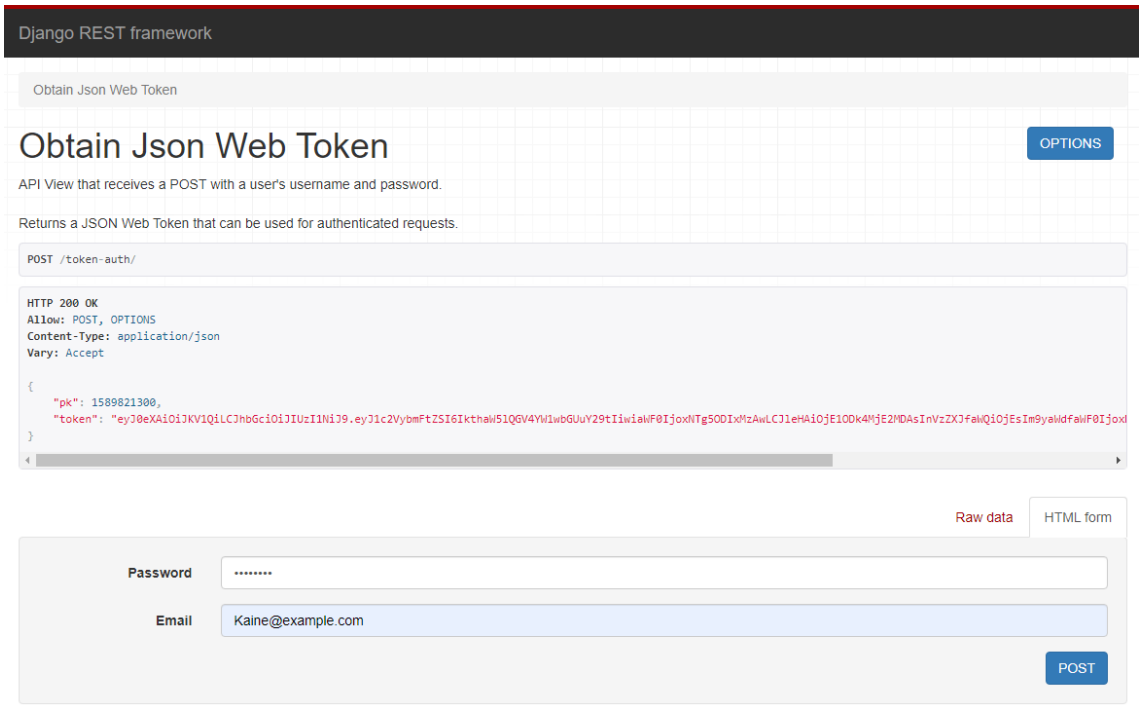

Figure 21. JWT token returned after login

The authentication API is ready, but user serializers are needed for the users to create and manage their profiles. Serializer in Django Rest Framework acts as a translator which can convert the Python data into data that be rendered into XML or JSON, or vice

versa.[17] Figure 22 and Figure 23 shows two user serializers which are used to create, update and view user accounts.

```
class UserSerializerWithToken(serializers.ModelSerializer):
    """Serializer for new user creation with token"""
    token = serializers.SerializerMethodField()
    password = serializers.CharField(write_only=True)

    def get_token(self, obj):
        jwt_payload_handler = api_settings.JWT_PAYLOAD_HANDLER
        jwt_encode_handler = api_settings.JWT_ENCODE_HANDLER

        payload = jwt_payload_handler(obj)
        token = jwt_encode_handler(payload)
        return token

    def create(self, validated_data):
        password = validated_data.pop('password', None)
        instance = self.Meta.model(**validated_data)
        if password is not None:
            instance.set_password(password)
        instance.save()
        return instance

    class Meta:
        model = get_user_model()
        fields = ('token', 'email', 'password', 'name')
        extra_kwargs = {'password': {'min_length': 5}}
```

Figure 22. UserSerializerWithToken for creating new users

Figure 22 describe user creation with a restriction of password (minimum length is 5 letters). During the creation, a manual token is generated with get_token function provided by the library. The token is then returned along with the new account.

```python
class UserSerializer(serializers.ModelSerializer):
    """Serializer for the user object"""

    class Meta:
        model = get_user_model()
        fields = ('email', 'password', 'name')

    def update(self, instance, validated_data):
        password = validated_data.pop('password', None)
        user = super().update(instance, validated_data)

        if password:
            user.set_password(password)
            user.save()

        return user
```

Figure 23. UserSerializer for viewing and updating users

The UserSerializer with view and update methods are kept separately because these methods are only for current authenticated user. The serializers are then connected to their respective views with different authentication requirements, as shown in Figure 24

```python
from rest_framework import generics, authentication, permissions

from user.serializers import UserSerializer, UserSerializerWithToken


class CreateUserView(generics.CreateAPIView):
    serializer_class = UserSerializerWithToken
    permission_classes = (permissions.AllowAny,)


class ManageUserView(generics.RetrieveUpdateAPIView):
    serializer_class = UserSerializer
    permission_classes = (permissions.IsAuthenticated,)

    def get_object(self):
        return self.request.user
```

Figure 24. Two user views in views.py

Both two views extend Django Rest Framework's generic views for faster implementation and concise code. CreateUserView provides a POST method, take the serializer class of UserSerializerWithToken and allows any users (unauthorized user included) to access the view. On the other hand, ManageUserView provides a GET, PUT and Patch method. It also opens to authenticated users only.[17] Routing the views to their URLs and making test requests grants desired results in Figure 25, 26 and 27.

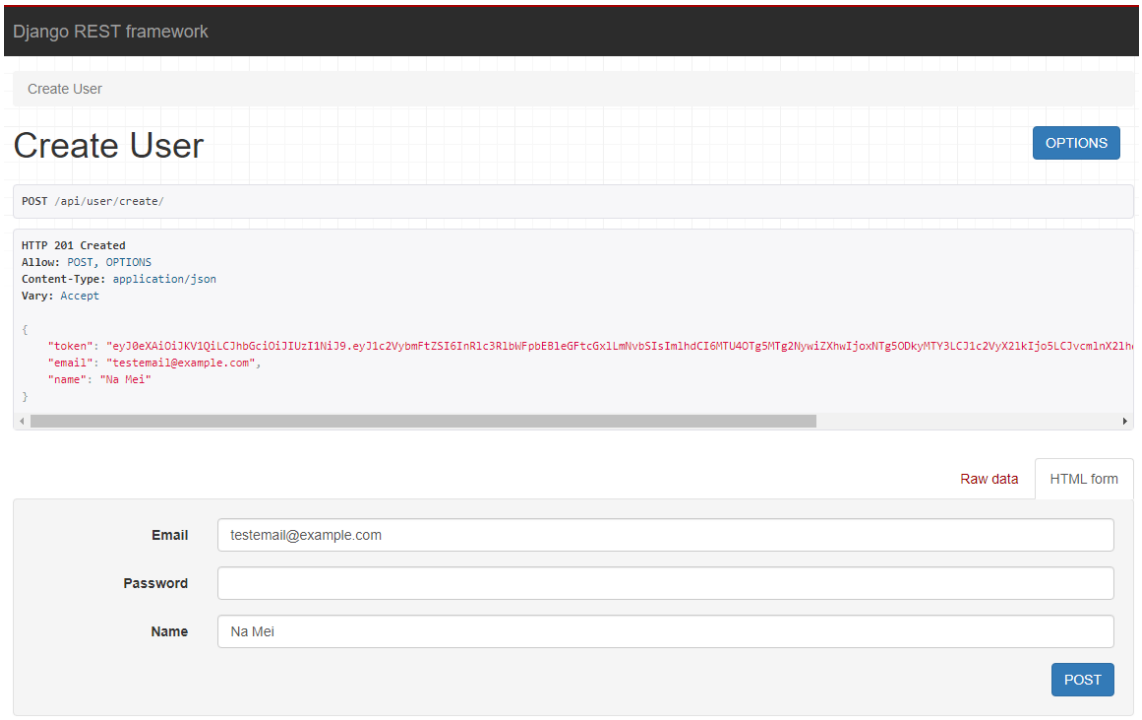

Figure 25. A HTTP 201 Created response from /api/user/create/

Figure 25 shows the response with token and user basic information from a POST request.

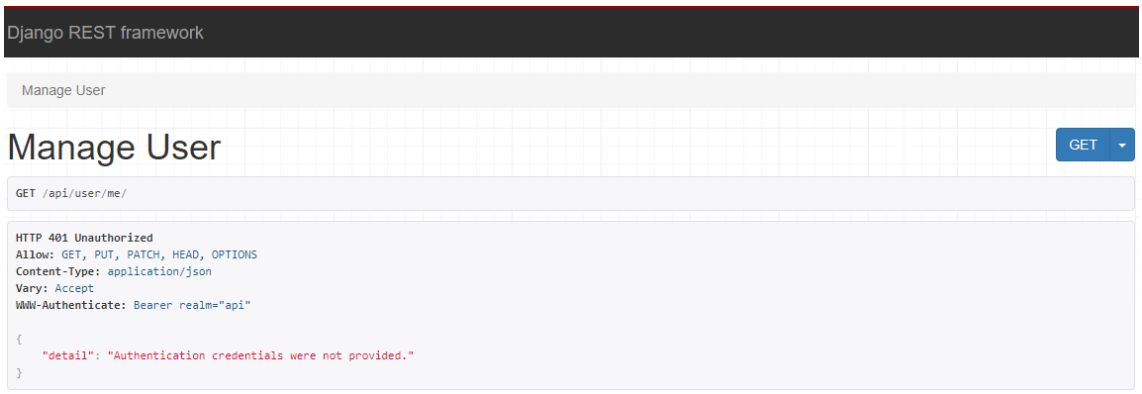

Figure 26. A HTTP 401 Unauthorized response from /api/user/me/

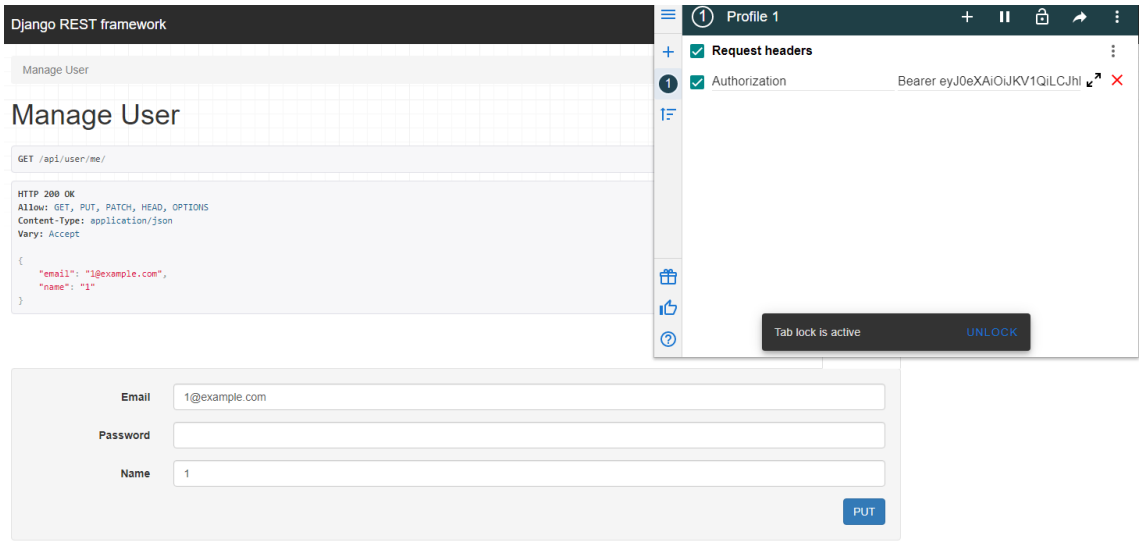

Figure 27. A HTTP 200 OK response from /api/user/me/

Figure 26 and 27 show different responses from the same url, depends on whether a user is authenticated or not. As seen from Figure 27, only the token is required for the request to be authenticated. This is a feature of JWT authentication, as stated in Django Rest Framework's documentation.[17]

On the whole, the user API is done with a combination of model in the database, serializers to customize the returned data and views to choose the serializer to present with customizations. Other models are done in a similar fashion without JWT.

4.3.3 Authentication on React

The authentication process on React revolves around four main actions: Sign up, Login, Logout and Resume session. Each of these actions are handled in its own function, which makes an API request to the server once called. The way these authentication actions work in React are listed below:

- Sign up: New user signs up for a new account and get the account information with the token back after the creation.
- Log in: A user log in using the email and password, then get the token back to store in the local browser's storage
- Log out: A user choose to log out of the system, the token inside local storage is automatically wiped.
- Resume session: On application starts, the local storage will be check if there is any token inside, then send a request along with the token to get the user account's information back.

Beside the resume session action, which is called when the application starts, other functions are action handlers for their respective components. The component for Login and Signup actions are mostly identical containing a form for the user to either log into the system or create a new account to access the system. Figure 28 contains the code of a Login form with the rendered view in Figure 29.

```
import React, {useState} from 'react';
import { makeStyles } from '@material-ui/core/styles';
import { Avatar, Button, CssBaseline,
    TextField, Typography, Container } from '@material-ui/core';

const LoginForm = props => {
  const classes = useStyles();
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const handle_email = e => {
    setEmail(e.target.value);
  }
  const handle_password = e => {
    setPassword(e.target.value);
  }
  const handle_submit = e => {
    props.handle_login(e, {"email": email, "password": password});
    setEmail("");
    setPassword("");
  }
  return (
    <Container component="main" maxWidth="xs">
      <CssBaseline />
      <div className={classes.paper}>
        <Avatar className={classes.avatar}>
        </Avatar>
        <Typography component="h1" variant="h5">
          Sign in
        </Typography>
        <form className={classes.form} onSubmit={handle_submit}>
          <TextField
            variant="outlined" margin="normal" required fullWidth
            id="email" label="Email Address" name="Email"
            autoComplete="email" autoFocus value={email} onChange={handle_email}
          />
          <TextField
            variant="outlined" margin="normal" required fullWidth
            name="Password" label="Password" type="password" id="password"
            autoComplete="current-password" value={password} onChange={handle_password}
          />
          <Button
            type="submit" fullWidth variant="contained"
            color="primary" className={classes.submit}
          >
            Sign In
          </Button>
        </form>
      </div>
    </Container>
```

Figure 28. Login form with Material UI

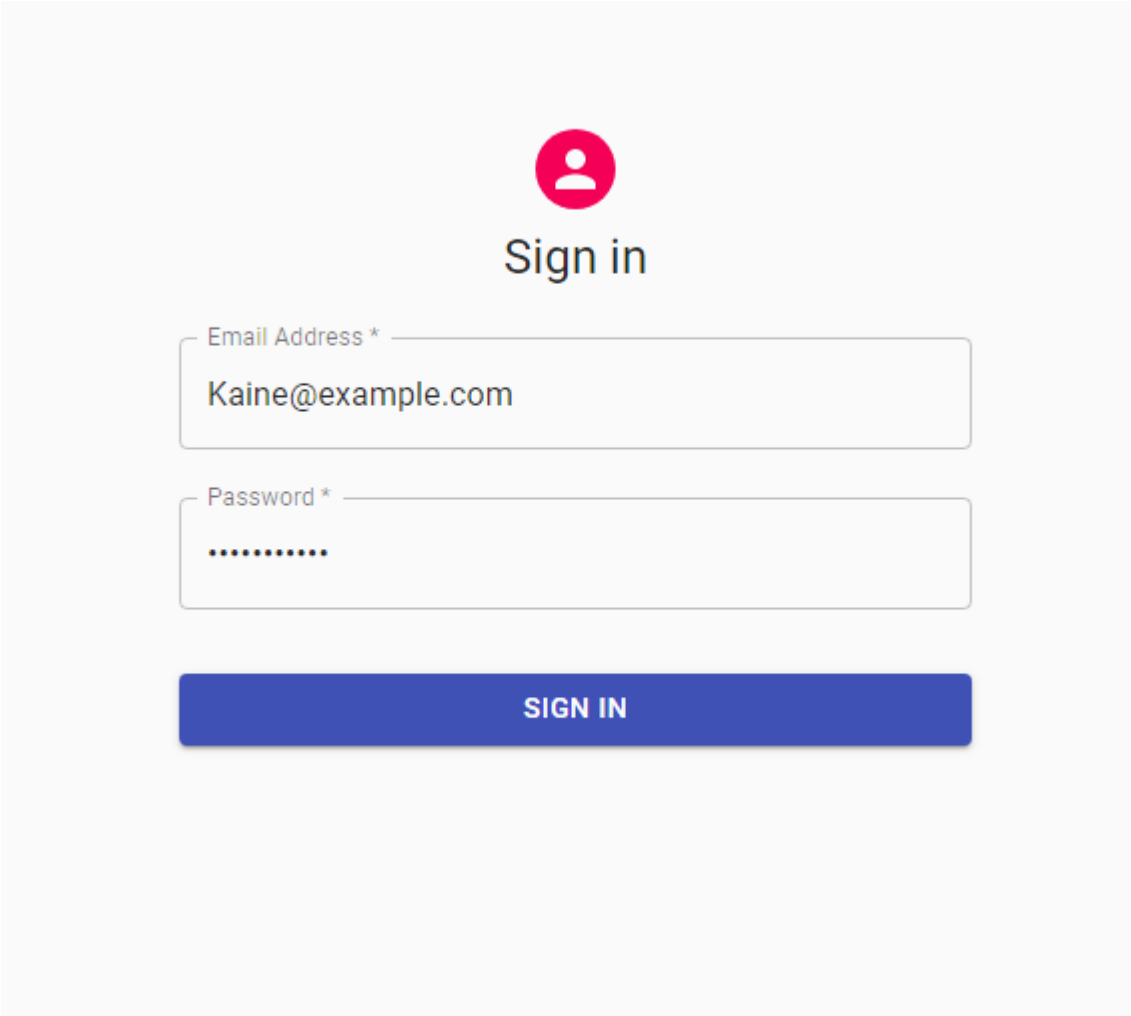

Figure 29. Sign in Form on the browser

The login form saves the state of the current typed email and password. It also triggers an action handler on form submit. This action handler makes an API call with data in the form, in this case the call returns a token and user's info as response. Then the states in the form will be deleted for security reasons

Since the authorization token and the user's info are needed for the whole application, they are stored as state in the main App component. A common practice for storing states for the whole application is to use a state management system such as Redux, MobX. However, considering the scale of the application, it is deemed unnecessary to implement an extra library into the system. With the main states remains the App component, the authentication actions should be defined in App as well due to React's one-

way data binding system. These functions are then passed as props to child components as action handlers. Figure 30 shows the login action handler in the App component.

```
const handle_login = (e,data) => {
  e.preventDefault();
  fetch('http://localhost:8000/token-auth/', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(data)
  })
  .then(res => res.json())
  .then(json => {
    localStorage.setItem('token', json.token);
    setUsername(username);
  })
}
```

Figure 30. Login action handler

Once logged in, the user is welcomed with the dashboard, showing the list of current requests with options of profile modification and logging out, as shown in Figure 31

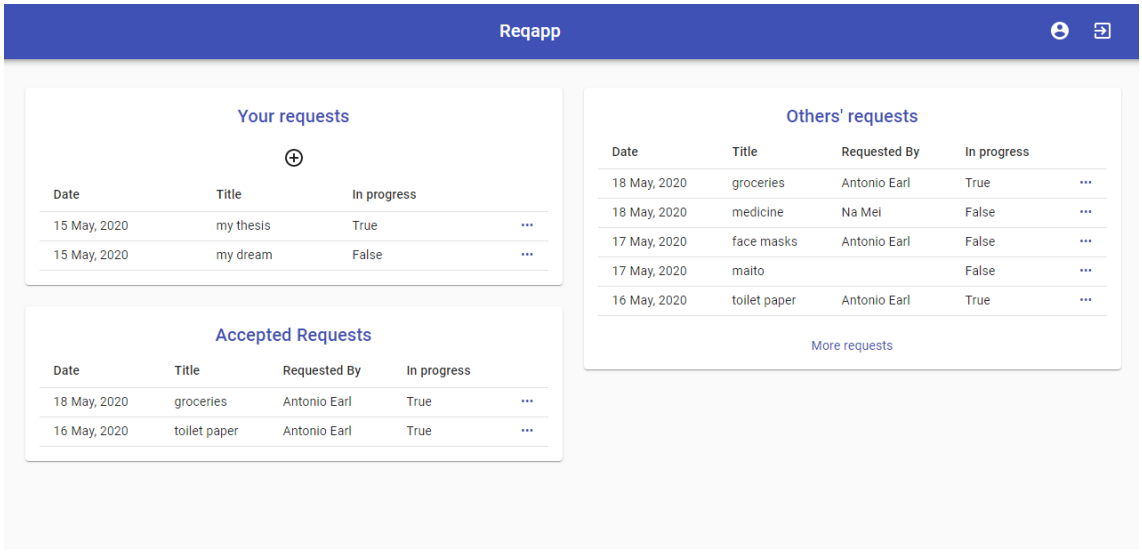

Figure 31. Application dashboard

In general, a React application consists of multiple components and interacts with the API through series of HTTP requests. Responses from these requests are then processed accordingly and stored in the states.

5 Discussions and Personal Thoughts

Since the scale of the project is small and the product itself is a prototype of a two-sided market, the data structure didn't have any problems and the implementation went smoothly. However, this is the first time the author has ever worked with any technologies beside JavaScript, the learning curve with the number of technologies involved in the project caused a large delay in development. Switching to only JavaScript with MERN stack would have been personally better choice for author considering the time constraint, the size of the project and the author's previous experience with JavaScript.

The second concern is the usage of Docker along with Travis CI, they were overlooked during the development process. Although The Docker container was created and run successfully, as well as a few testing suites written for Travis CI, the involvements didn't cover the whole application and leave a lot to be desired. This can be solved with better time management and experience.

That being said, the effort spent in learning Django with Docker are worth it. Django and Django Rest Framework are well-customized web frameworks offering a lot of built-in functionality. While the documentations are extensive and rather overwhelming at first, the amount of required coding is way less than in building a backend with Node. Database migrations and API routing process was painlessly done, thanks to the serializers and class inheritances. Comparing to Node, Django seems to be more scalable, while Node is a bit better performance-wise. The study in Docker is also interesting, it solves a lot of compatibility problems when deploying the app to multiple platforms.

6 Conclusion

The goal of the thesis was to build a prototype of a two-sided market and study full-stack web development with React and Django. The concepts of web development and related technologies were heavily discussed, followed by example of a feature implementation process with in-depth explanation.

The final prototype was a two-sided market for creating and fulfill requests during quarantine time. Users were able to create a new account, manage their profile, make new requests and accept other requests. The API was designed to grant access to authenticated users. Unauthorized users were only allowed on sign up and login page.

Overall, the project was a sufficient showcase of how full-stack web development can be done with Django and React. Once deployed the application would be able to let people help each other during pandemic times. It has the potential for further development with more complex data structure and features such as location-based requests list or built-in messenger for better contact.

References

1 Q&A on Coronaviruses (COVID-19). 2020. WHO. Web document. https://www.who.int/emergencies/diseases/novel-coronavirus-2019/question-and-answers-hub/q-a-detail/q-a-coronaviruses#:~:text=symptoms

2 What is a web application? 2019. Daniel Nations. Web document. https://www.lifewire.com/what-is-a-web-application-3486637

3 Web Application Architecture: Definition, Models, Types, and More. 2020. Swapnil Banga. Web document. https://hackr.io/blog/web-application-architecture-definition-models-types-and-more

4 A Beginner's Guide to Front-End Development. 2017. Carey Wodehouse. Web document. https://www.upwork.com/hiring/development/beginners-guide-to-front-end-development/

5 What Is the Difference Between Front-End and Back-End Development? 2019. Concepta. Web document. https://www.conceptainc.com/blog/difference-front-end-back-end-development

6 LAMP Stack. 2019. IBM Cloud Education. Web document. https://www.ibm.com/cloud/learn/lamp-stack-explained

7 The Modern Application Stack – Part 1: Introducing the MEAN Stack. 2017. Andrew Morgan. Web document. https://www.mongodb.com/blog/post/the-modern-application-stack-part-1-introducing-the-mean-stack

8 React. 2020. React. Web document. https://reactjs.org/

9 Stack Overflow Developer Survey 2019 Insights. 2019. Stack Overflow. Web document. https://insights.stackoverflow.com/survey/2019#technology

10 Interactive React Lifecycle Methods diagram. 2020. Web document. https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

11 React Virtual DOM Explained in Simple English. 2018. Mosh Hamedani. Web document. https://programmingwithmosh.com/react/react-virtual-dom-explained/

12 Django Introduction. 2020. Django. Web document. https://www.djangoproject.com/

13 Django design philosophies. 2020. Django. Web document. https://docs.djangoproject.com/en/3.0/misc/design-philosophies/

14 Django FAQ. 2020. Django. Web document. https://docs.djangoproject.com/en/3.0/faq/general/

15 MVC Architecture. 2020. Google. Web document. https://developer.chrome.com/apps/app_frameworks

16 Django's Structure – A Heretic's Eye View. 2020. The Django Book. Web document. https://djangobook.com/mdj2-django-structure/

17 Django Rest Framework. 2020. Django Rest Framework. Web document. https://www.django-rest-framework.org/

18 About PostgreSQL. 2020. PostgreSQL. Web document. https://www.postgresql.org/about/

19 Writing your first Django app, part 2. 2020. Django. Web document. https://docs.djangoproject.com/en/3.0/intro/tutorial02/

20 venv - Creation of virtual environments. 2020. Python. Web document. https://docs.python.org/3/library/venv.html

21 What is a container. 2020. Docker. Web document. https://www.docker.com/resources/what-container

22 Pro Git. 2014. Scott Chacon, Ben Straub. Web document. https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

23 Core Concepts for Beginners. 2020. TravisCI. Web document. https://docs.travis-ci.com/user/for-beginners/

24 QuickStart: Compose and Django. 2020. Web document. https://docs.docker.com/compose/django/

25 Create-react-app documentation. 2020. Web document. https://create-react-app.dev/docs/getting-started/

26 Tutorial: Django REST with React (Django 3 and a sprinkle of testing). 2020. Valentino Gagliardi. Web document. https://www.valentinog.com/blog/drf/

27 REST framework JWT Auth documentation. 2020. Web document. https://styria-digital.github.io/django-rest-framework-jwt/